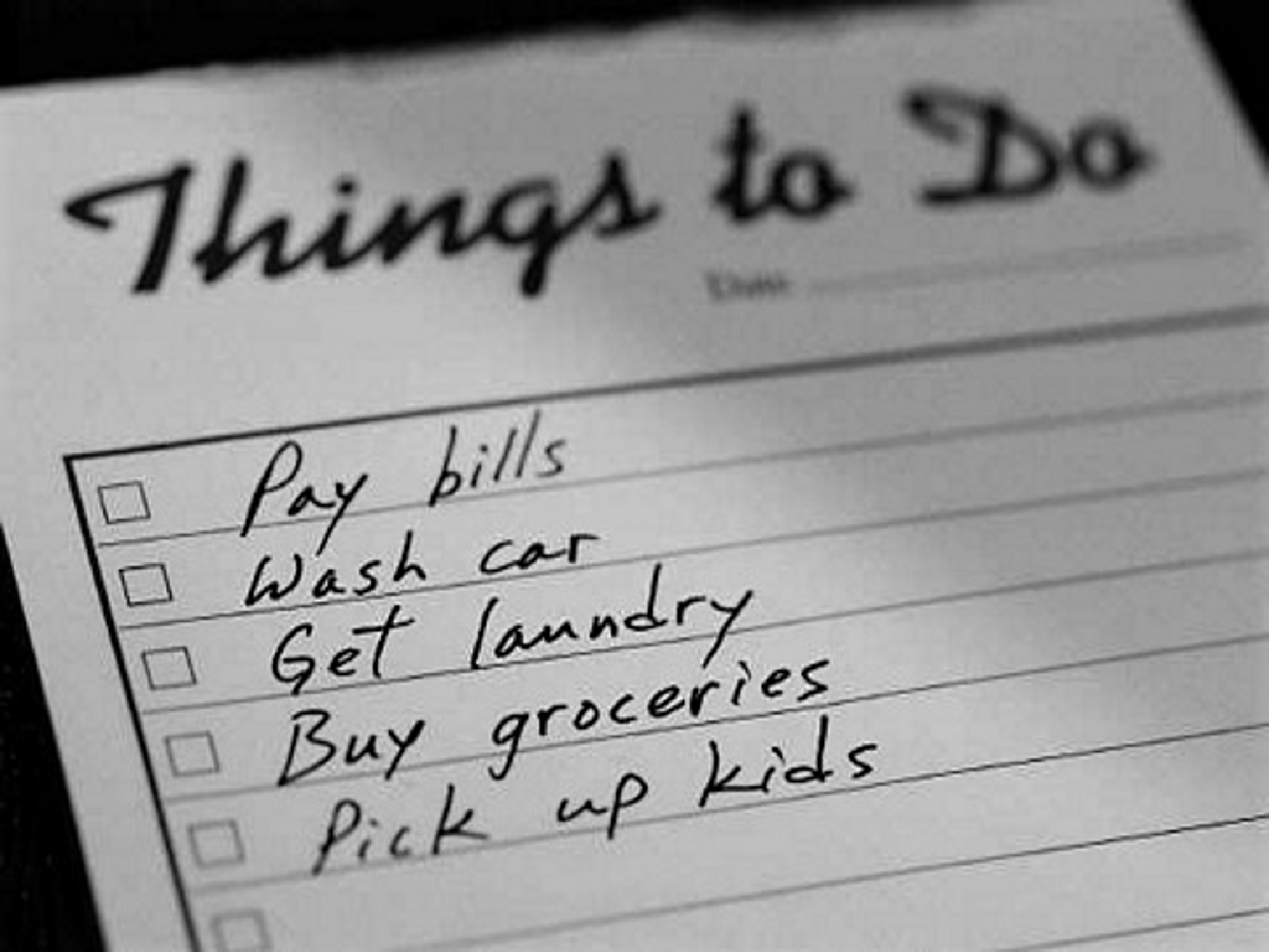
Lists



Things to Do

Date

- ☐ Pay bills
- ☐ Wash car
- ☐ Get laundry
- ☐ Buy groceries
- ☐ Pick up kids
- ☐

Lists

A *list* is like an array, except that the length changes

Items are added to a list over time, and you don't know in advance how many there will be

This chapter implements lists in two ways in C

The same ideas apply to lists in almost every other programming language

Two approaches

There are two approaches that you can take:

- *array lists*
- *linked lists*

Array lists are better when the most common operations are to add or remove at one end

Linked lists are better when frequently inserting and deleting in the middle

Array list of int

Suppose we want a list of `ints`

We can use a flexible array, with a length variable to say how full it is:

```
int length = 0, capacity = 4;
int *items = malloc(capacity * sizeof(int));
...
if (length >= capacity) {
  capacity = capacity * 3 / 2;
  items = realloc(items, capacity * sizeof(int));
}
...
```

First attempt

It seems as though we need an add function, for adding an int to the list, like this:

```
void add(int length, int capacity, int *items, int n) {
...
```

We would need to pass the length and capacity as well as the array and new item

Failure

The function doesn't work because it can't update the caller's `length` variable, the caller's `capacity` variable, or the caller's `items` variable in case the array is moved by `realloc`

Even if the function did work, it is tiresome passing around the three variables separately

Second attempt

We could pass in all three variables in one go, and pass the updated versions of those three variables back as a result:

```
struct list { int length, capacity, *items; };
typedef struct list list;

list add(list ns, int n) {
    ...
}
...
ns = add(ns, 42);
```

Poor attempt

The second attempt does work, but has two flaws

- it is too easy to make the mistake of writing `add(ns,n)` instead of `ns = add(ns,n)`
- if a function needs to return some other result, as well as the updated list, we are stuck again

Third attempt

How do we achieve really simple function calls like `add(ns,n)` ?

Answer, pass the list structure by pointer:

```
struct list { int length, capacity, *items; };
typedef struct list list;

void add(list *ns, int n) {
    ...
}
```

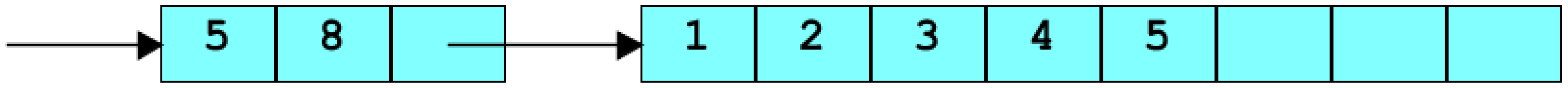
This treats a list as an 'object'

Picture

We now need a good picture of the situation

```
struct list { int length, capacity, *items; };
typedef struct list list;

list *ns;
```



Pointer purposes

The pointers in the picture have two different purposes

The first, `ns`, allows functions to update the list structure in place

The second, `items`, allows the array to be moved and resized

Array list demo program

To make a demo program, it is useful to write `main` first, to see what we want the function calls to look like:

```
int main() {                                arraylist.c
    list *numbers;
    numbers = newList();
    add(numbers, 3);
    add(numbers, 5);
    add(numbers, 42);
    print(numbers);
}
```

The functions that make this possible have been kept small, and designed as if they were library functions, to keep everything under control

A creation function

Here's a function to make a new list:

```c
// Make a new empty list                          arraylist.c
list *newList() {
    list *ns = malloc(sizeof(list));
    int *items = malloc(4 * sizeof(int));
    *ns = (list) { 0, 4, items };
    return ns;
}
```

An expand function

Here's a function to expand a list:

```
// Make a list bigger                                  arraylist.c
void expand(list *ns) {
    ns->capacity = ns->capacity * 3 / 2;
    ns->items = realloc(ns->items, ns->capacity);
}
```

An add function

Here's a function to add an int to the list:

```
// Add an int to a list                          arraylist.c
void add(list *ns, int n) {
    if (ns->length >= ns->capacity) expand(ns);
    ns->items[ns->length] = n;
    ns->length++;
}
```

A print function

Here's a function to print the list:

arraylist.c

```
// Print a list
void print(list *ns) {
    for (int i=0; i<ns->length; i++) {
        if (i > 0) printf(", ");
        printf("%d", ns->items[i]);
    }
    printf("\n");
}
```

Testing

How would you test a list module like this?

You need to use all your programming skills

- creatively design the testing
- improve the design as you go along
- use small functions and small amounts of progress
- develop tests ***alongside*** the functions

Setup

Here's how the testing might end up

First a function to set up an example list (made of integers 0 to 9)

```
static list *setup(char *digits) {
    list *ns = newList();
    for (int i = 0; i < strlen(digits); i++) {
        int n = digits[i] - '0';
        add(ns, n);
    }
}
```

Call

Next a function to call an operation by name

```
static int call(list *ns, char *op, int arg) {
    int result = 0;
    if (strcmp(op, "add") == 0) add(ns, arg);
    if (strcmp(op, "get") == 0) result = get(ns, arg);
    if (strcmp(op, "expand") == 0) expand(ns);
    ...
    return result;
}
```

Imagine there is a `get` function to find the n'th item in a list

Check

Next a function to check the contents of the list after the operation

```c
static bool check(list *ns, char *digits) {
    if (ns->length != strlen(digits)) return false;
    for (int i = 0; i < strlen(digits); i++) {
        int n = digits[i] - '0';
        if (get(ns, i) != n) return false;
    }
    return true;
}
```

Run test

Next a function to run a given test

```
static bool test(char *op, char *pre, char *in,
        char *post, char *out) {
    list *ns = setup(pre);
    int arg = in[0] == '\0' ? 0 : in[0] - '0';
    int ret = out[0] == '\0' ? 0 : out[0] - '0';
    int result = call(ns, op, arg);
    return result == ret && check(ns, post);
}
```

A brand new list is set up for every test, so that tests can't affect each other

The tests

Finally a function to carry out the tests

```
int listMain() {
    assert(test("add", "1234", "5", "12345", ""));
    assert(test("get", "1234", "2", "1234", "3"));
    assert(test("expand", "1234", "", "1234", ""))
    ...
}
```

Templates

Each of the functions costs some effort to write

By keeping them short and developing them one at a time, it is possible to keep everything under control

The outcome is worth it, for the simplicity of the calls

The functions only work on `int`s

But in another project, even if a list of some other type is needed, these functions will make good templates

Array lists of structures

Suppose we want an array list of structs

We can copy the `int` functions, and change `int` to (say) `struct point` (maybe using a typedef)

Then the `items` field in the list structure would be an array of raw structures

It is a pity C doesn't (convincingly) support "list of anything", and we have to rename one set of functions if we want to use both in one program

Big structures

What if the structs are big?

Then there are two problems

The fact that the array is not full means that there is a large amount of wasted space because of the unfilled structures

More important, perhaps, is that the structures will get copied into the list, instead of being shared with versions held in other places (and updates to the originals will not be reflected in the copies)

Object lists

That suggests that we use our functions as a template still, but replacing `int` with `struct x *`, i.e. we store a list of pointers to structures

This now means that our list has three layers of pointers: a pointer to the list structure, a pointer from there to the array, and then the array consists of pointers to the item structures

The complexity can be worth it, and it is what is done in object oriented languages (with the pointers being automated)

Linked lists

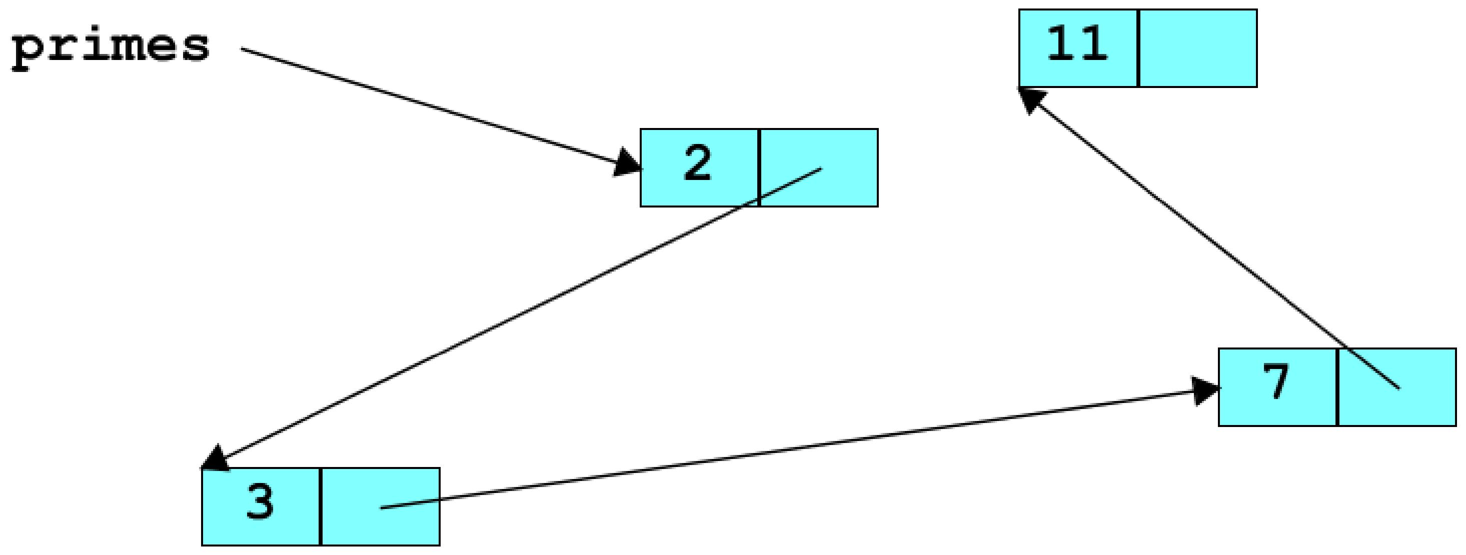
A problem with array lists is that, to insert or delete an item in the middle, lots of items have to be moved up or down to make space

Can we find a way of storing a list so that items never have to be moved?

One way is to introduce a pointer to go with each item, pointing to the next item

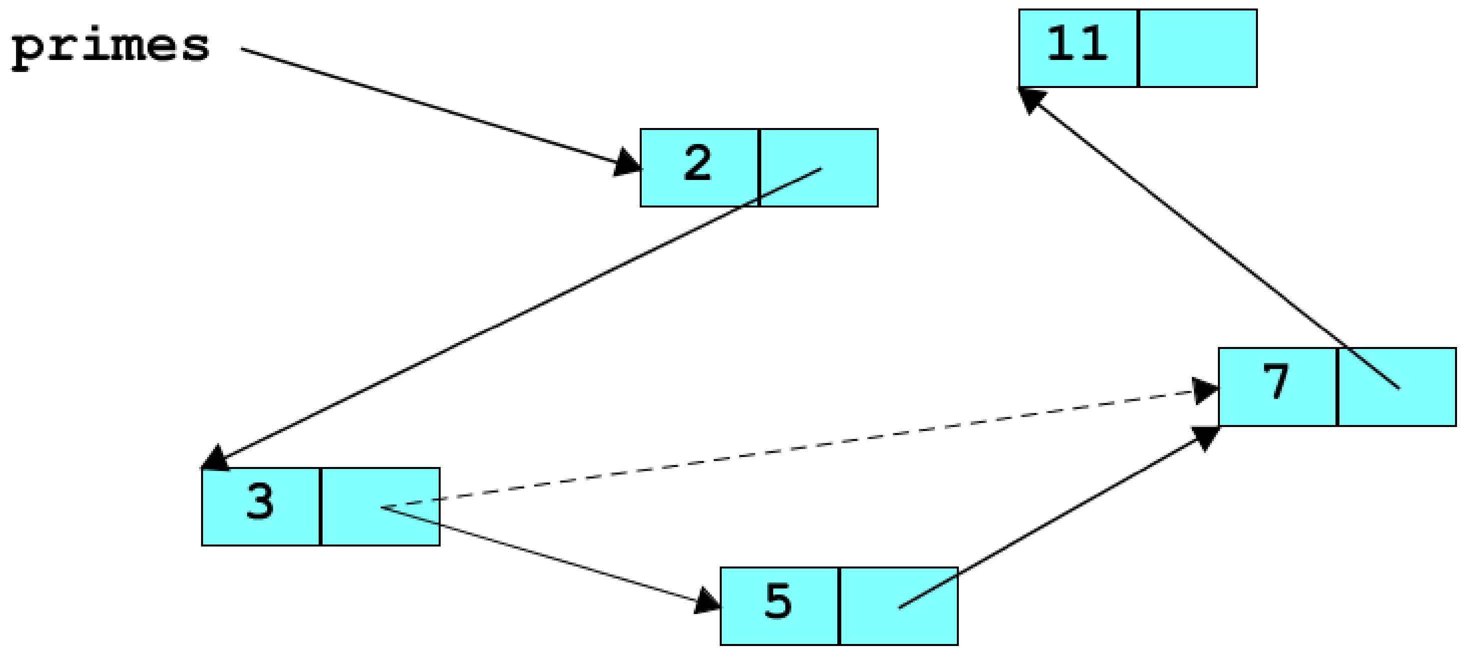
Example: primes

A linked list of primes (without 5) might look like this



Example: insertion

After inserting 5, it might look like this



Insertion

To insert 5 into the list, these steps are needed:

- find the structures containing 3 and 7
- allocate some space for a new structure
- set the first field to 5
- set the second field to point to the 7 structure
- change 3's pointer to point to the new structure

The list entries end up scattered in memory, but it doesn't matter where they are

Stack

The easy and efficient operations on a linked list are called the ***stack*** operations:

- ***push***: insert an item at the start of the list
- ***pop***: remove an item from the start of the list
- ***top***: look at the first item (sometimes called peek)
- ***isEmpty***: check if there are any items

Design problem

As before, with lists, we have a design problem

Suppose a stack is just a pointer to the first item

Then the push and pop operations need to change the stack variable

With push, we could use `stack = push(stack,n)`, catching the returned updated list

But we want `pop` to return the first item - it can't easily also return the updated list

So let's have a separate list structure as well, like before

Stack demo

Here's the main function of a stack demo, so we can see what the function calls look like:

stack.c

```
int main() {
    list *stack;
    stack = newStack();
    push(stack, 3);
    push(stack, 5);
    push(stack, 7);
    printf("top %d\n", top(stack));
    while (! isEmpty(stack)) {
        int n = pop(stack);
        printf("%d\n", n);
    }
}
```

Stack structures

The structures needed for the stack demo are:

stack.c

```
struct cell {
    int item;
    struct cell *next;
};

struct list {
    struct cell *first;
};
```

The first is for the individual items, the second is for the list as a whole (and you can add the usual typedefs)

New stack

The function to create a new stack is:

stack.c

```
list *newStack() {
    list *new = malloc(sizeof(list));
    new->first = NULL;
    return new;
}
```

`NULL` is a special pointer which doesn't point anywhere - it is used for empty lists, and in the last cell in a list

(It is usually defined as address `0`, a part of memory which never belongs to your program, so it causes a segmentation fault crash if you follow it)

Check stack empty

The function to check if a stack is empty is:

stack.c

```
bool isEmpty(list *stack) {
    return stack->first == NULL;
}
```

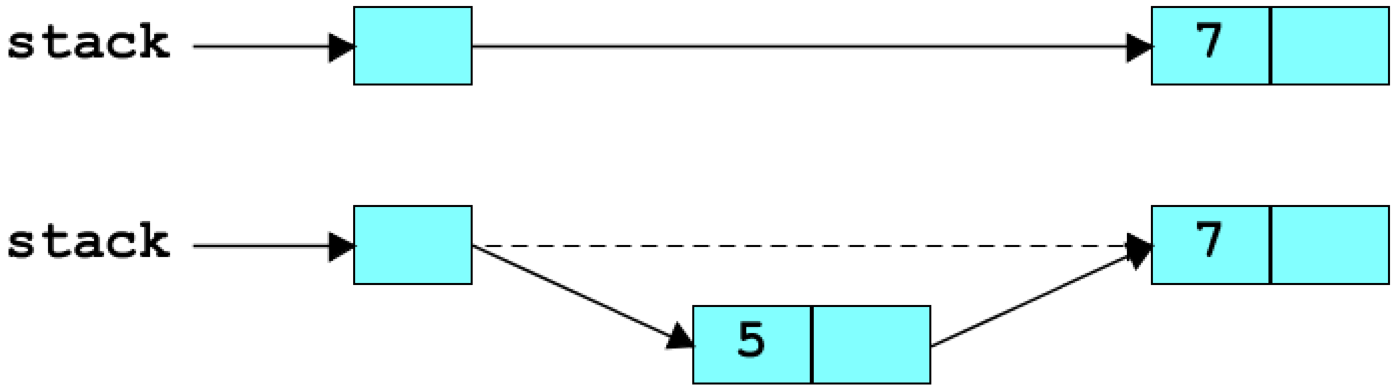
Push onto stack

The function to push an item onto a stack is:

stack.c

```
void push(list *stack, int n) {
    cell *new = malloc(sizeof(cell));
    *new = (cell) { n, stack->first };
    stack->first = new;
}
```

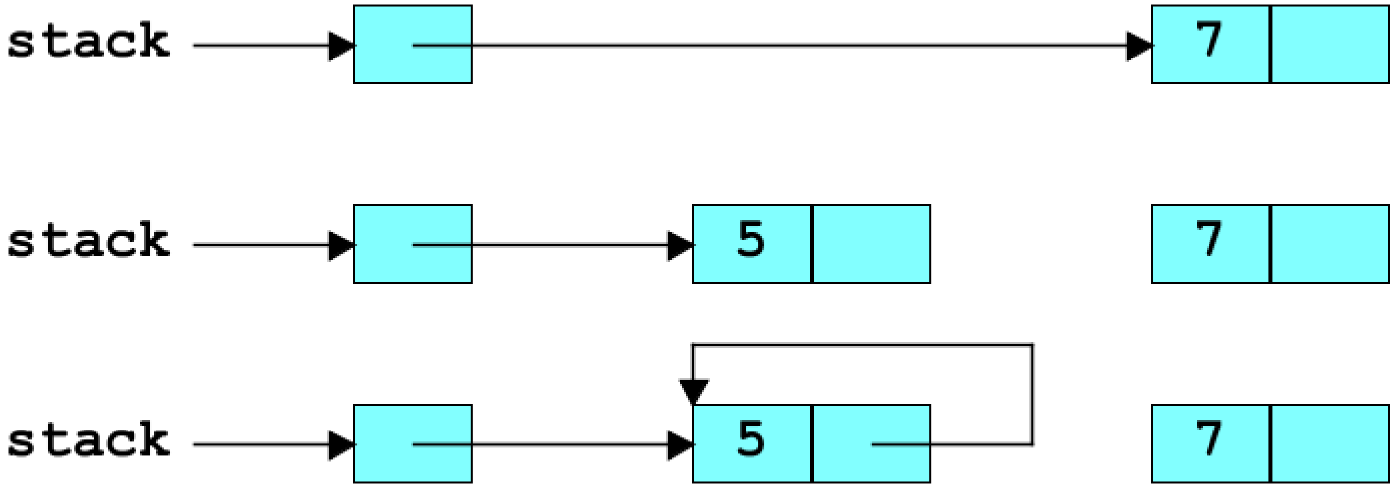
Before and after picture:



Classic mistake

A very common mistake with pointer handling is to do things in the wrong order:

```
void push(list *stack, int n) {                          stack.c
    cell *new = malloc(sizeof(cell));
    stack->first = new;                             // BAD CODE
    *new = (cell) { n, stack->first };
}
```



Stack errors

It is a good idea to have a function to call if something goes disastrously wrong:

```
void fail(char *message) {                        stack.c
    fprintf(stderr, "%s\n", message);
    exit(1);
}
```

The function prints to `stderr`, and stops the program with an error code (as if returning `1` from `main`) to play nicely with any scripts that include the program

Top of stack

The function to look at the top item is:

```
int top(list *stack) {                                                stack.c
    if (stack->first == NULL) fail("top of empty stack");
    return stack->first->item;
}
```

If the caller tries to get the top item from an empty stack, the `fail` function is called

Otherwise, the program might do rubbish things

Pop from stack

The function to remove the top item is:

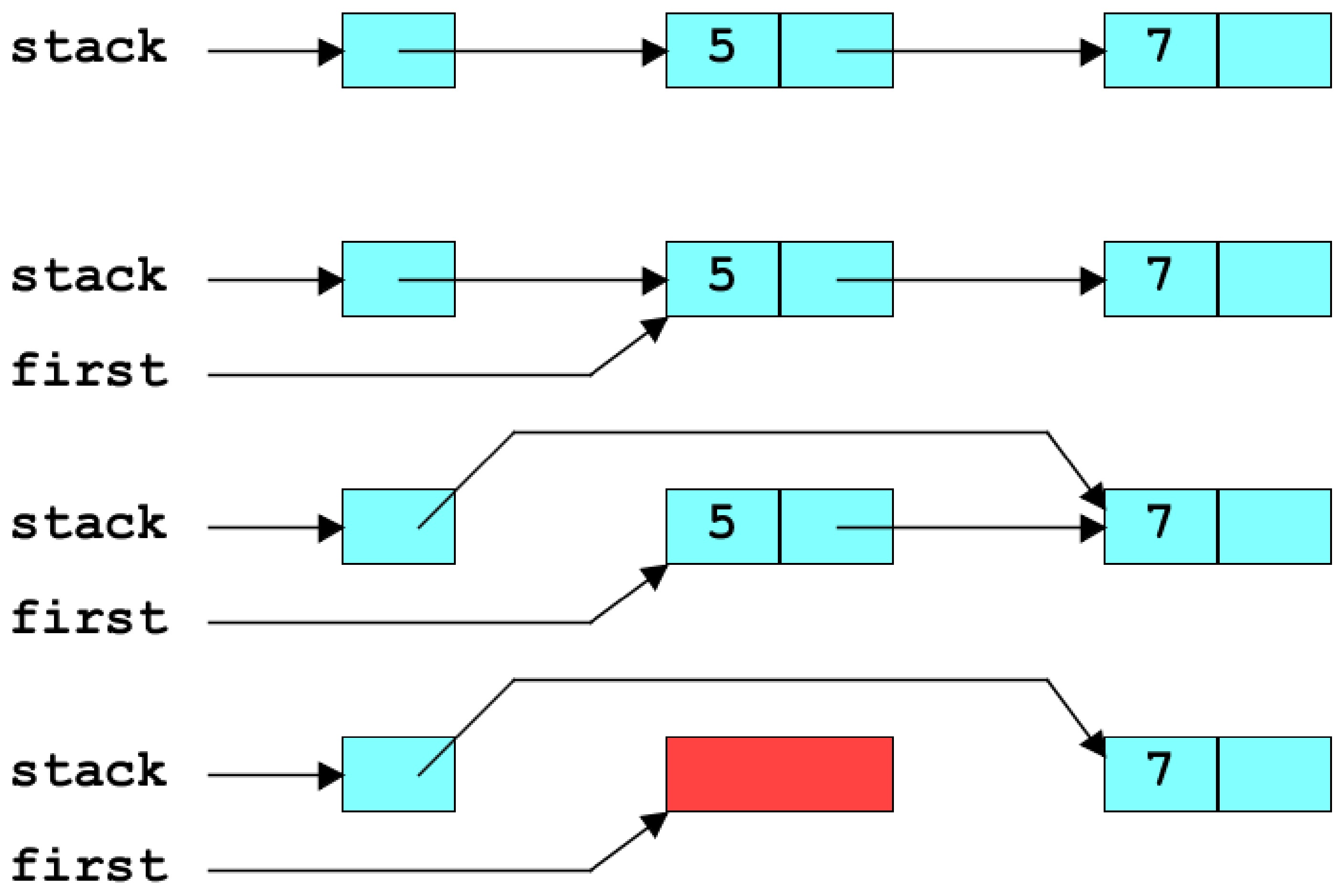
stack.c

```
int pop(list *stack) {
    cell *first = stack->first;
    if (first == NULL) fail("pop of empty stack");
    stack->first = first->next;
    int n = first->item;
    free(first);
    return n;
}
```

This has to be written incredibly carefully, saving the first cell in a variable before removing it from the list, and extracting its fields before freeing up its space

Visualising pop

The main steps in pop are:



Structure lists

To store structures instead of ints, you could include the `next` field in the structure, e.g.

```
struct cell {
    char *name;
    int number;
    struct cell *next;
};
```

The `next` field can be ignored everywhere except in the list functions

Although this is common in tutorials etc., it doesn't allow an item to be stored in more than one list

Object lists

A more flexible approach is to store objects, i.e. pointers to structures, in lists:

```
struct cell {
    struct entry *item;
    struct cell *next;
};
```

This has an extra layer of pointers, but now an object can appear in any number of lists, and updates to objects are shared by all occurrences

Efficiency

There is an efficiency problem with what we have done

All the stack functions are supposed to be O(1), but they may not be

That is because of the cost of `malloc` and `free` which can, at worst, have O(n) behaviour

Free list

To overcome the problem, it is common for a list structure to contain a *free list*, i.e. a list (stack) of cells which are currently unused but are free to be re-used

```
struct list {
    struct cell *first;
    struct cell *free;
};
```

You put cells on the free list instead of calling `free`

And when you want a new cell, you get it from the free list if possible, and only allocate a new one if the free list is empty

Modules

Once you have built a good implementation of stacks, it is natural to re-use it in other programs

To do that, you put the stack functions into a separate module

And you make sure that programs cannot access the cells being used, and in fact cannot tell how the stack is being implemented - it is just a service, and a robust one

List variations

- keep track of the last cell in the list structure, to allow adding at the end
- keep track of the length of the list
- keep track of a current position within the list, to allow traversal and insertion in the middle
- keep track of the cell before the current one in the list, to allow deletion of the current item
- have a previous pointer as well as a next pointer in each cell, to make deletions easier
- have dummy cells which go before the first one and after the last, to simplify the code by getting rid of `NULL` tests

The real truth

For implementing general lists, and stacks, an array list is almost always better than a linked list

Linked lists use a lot of memory, the efficiency of insertion and deletion in the middle is offset by the cost of finding or keeping track of the right place in the list, and they are difficult to encapsulate well

But the idea of linked lists comes up a lot in computer science, they are often used as part of something else (e.g. hash tables), and variations are often used (e.g. a linked list in an array)